

Prohledávání stavového prostoru

Aleš Horák

E-mail: hales@fi.muni.cz
<http://nlp.fi.muni.cz/uui/>

Obsah:

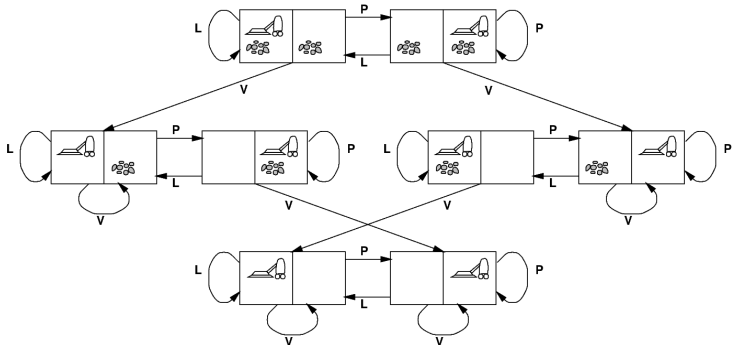
- Prohledávání stavového prostoru
- Neinformované prohledávání
- Informované prohledávání stavového prostoru
- Jak najít dobrou heuristiku?

Prohledávání stavového prostoru

Řešení problému prohledáváním stavového prostoru:

- **stavový prostor**, předpoklady – statické a deterministické prostředí, diskrétní stavy
- *počáteční stav* **problem.init_state**
- *cílová podmínka* **problem.is_goal(State)**
- *přechodové akce* **problem.moves(State) → NewStates**
- implementace a zpracování výstupů **problem.moves()** určují **prohledávací strategii**

Problém agenta Vysavače

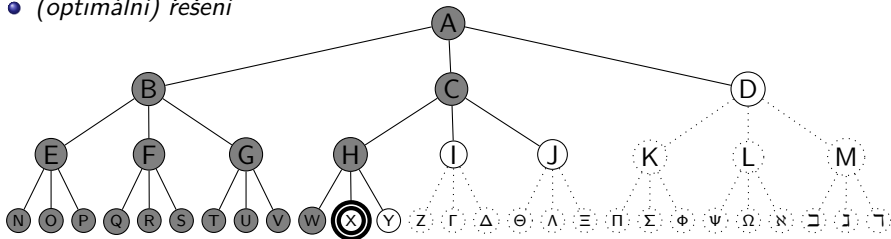


- máme dvě **místnosti** (L, P)
- jeden **vysavač** (v L nebo P)
- v každé místnosti je/není špína
- počet **stavů** je $2 \times 2^2 = 8$
- **akce** = {doLeva, doPrava, Vysávej}

Problém agenta Vysavače

Prohledávací strategie – prohledávací strom:

- kořenový uzel
- uzel prohledávacího stromu:
 - (odkaz na) stav
 - rodičovský uzel
 - přechodová akce
 - hloubka uzlu
 - cena – $g(n)$ cesty, $c(x, a, y)$ přechodu
- (optimální) řešení




Problém agenta Vysavače

Prohledávací strategie – prohledávací strom:

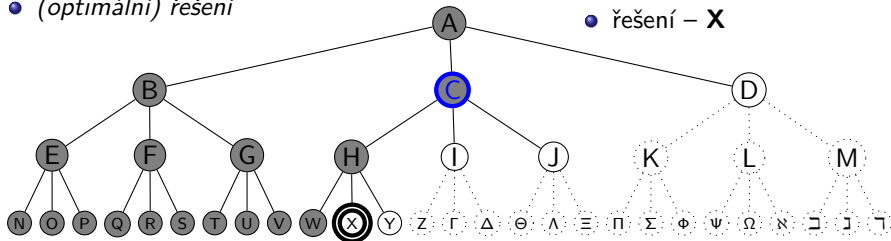
- kořenový uzel
- uzel prohledávacího stromu:
 - (odkaz na) stav
 - rodičovský uzel
 - přechodová akce
 - hloubka uzlu
 - cena – $g(n)$ cesty, $c(x, a, y)$ přechodu
- (optimální) řešení

- A** (stav )

- např. uzel **C**:

- stav – 
- rodič – **A**
- akce – **doPrava**
- hloubka – **1**
- cena – **1**

- řešení – **X**



Řešení problému prohledáváním

Kostra algoritmu:

```
function SEARCH(problem)  
  process ← collection(problem.init_state) # stavy ke zpracování, blíže neurčená kolekce  
  while length(process) > 0 do  
    current_node ← remove_current_node(process)  
    if problem.is_goal(current_node) then print current_node # řešení  
    foreach child in problem.moves(current_node) do  
      process.add_node(child)
```

Řešení problému prohledáváním

Kostra algoritmu:

```
function SEARCH(problem)
  process ← collection(problem.init_state) # stavy ke zpracování, blíže neurčená kolekce
  while length(process) > 0 do
    current_node ← remove_current_node(process)
    if problem.is_goal(current_node) then print current_node # řešení
    foreach child in problem.moves(current_node) do
      process.add_node(child)
```

rekurzivně včetně cesty k řešení:

```
function RECURSIVESEARCH(problem, path = collection())
  if length(path) = 0 then
    return RecursiveSearch(problem, collection(problem.init_state))
  current_node = get_current_node(path)
  if problem.is_goal(current_node) then print path # cesta k řešení
  foreach child in problem.moves(current_node) do
    RecursiveSearch(problem, path.with_node(child)) # cesta rozšířená o child
```

Prohledávací strategie – vlastnosti

Porovnání strategií:

- úplnost
- optimálnost
- časová složitost
- prostorová složitost

Prohledávací strategie – vlastnosti

Porovnání strategií:

- úplnost
 - optimálnost
 - časová složitost
 - prostorová složitost
- složitost závisí na:
- b – faktor **větvení** (branching factor)
 - d – hloubka **cíle** (goal depth)
 - m – **maximální hloubka** větve/délka cesty (maximum depth/path, může být ∞ ?)

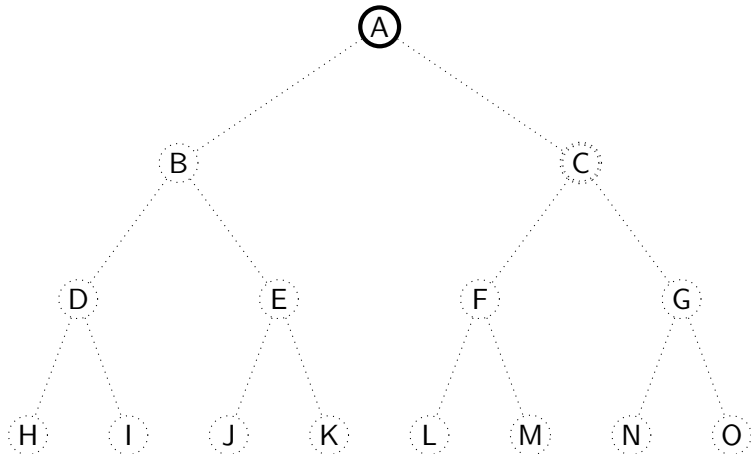
Neinformované prohledávání

- prohledávání do hloubky
- prohledávání do hloubky s limitem
- prohledávání do šířky
- prohledávání podle ceny
- prohledávání s postupným prohlubováním

poznámka – zde názvy metod označují **prohledávací strategie**, které **nejsou totožné** s odpovídajícími obecnými grafovými algoritmy

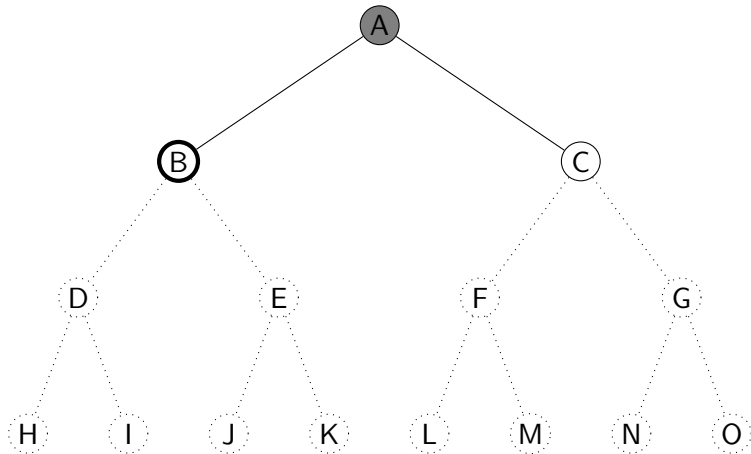
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



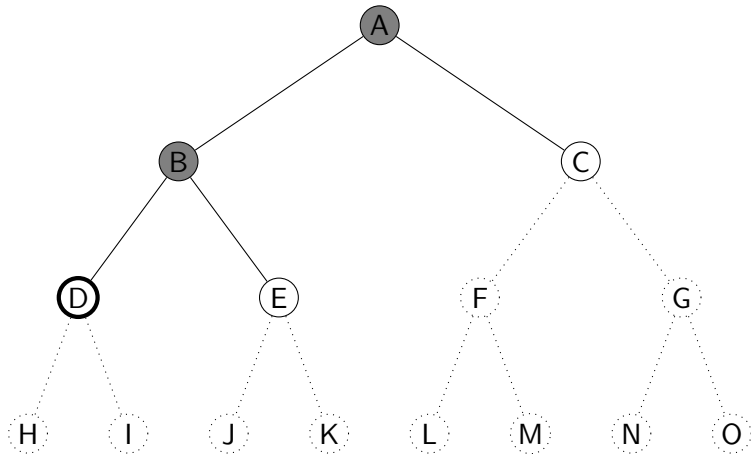
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



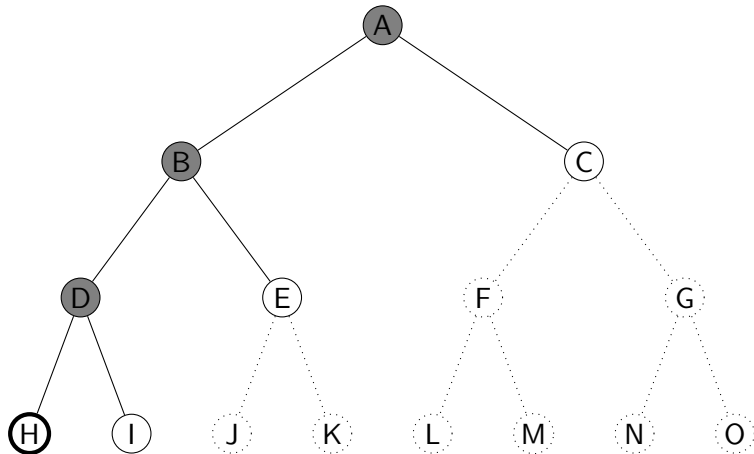
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



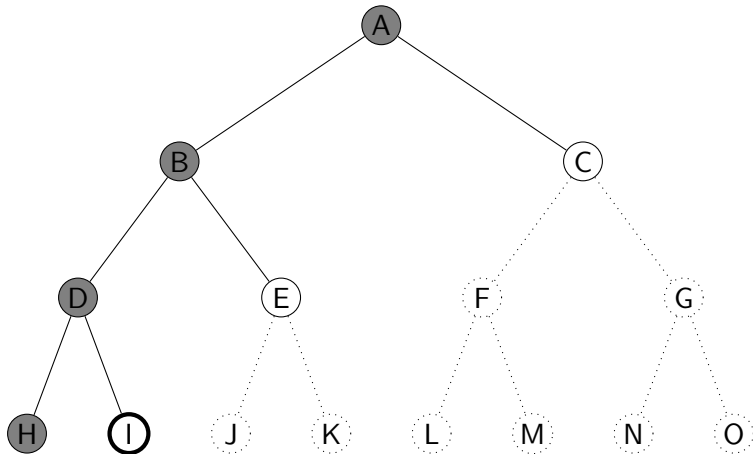
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



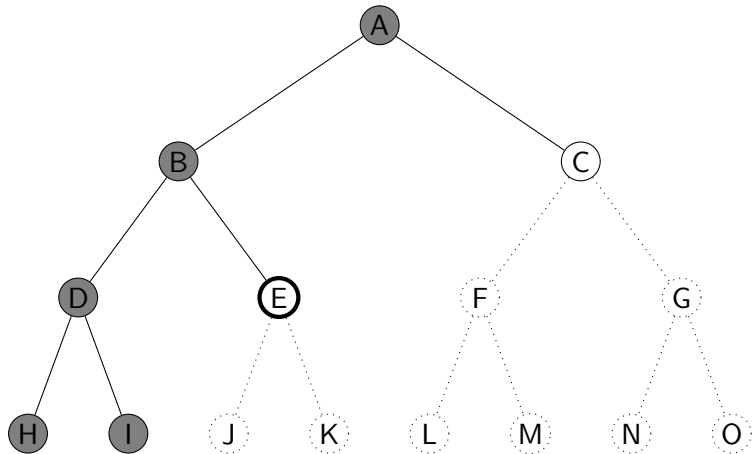
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



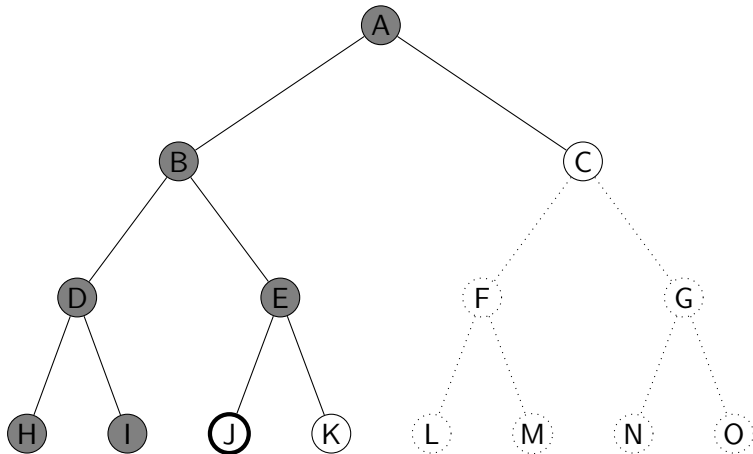
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



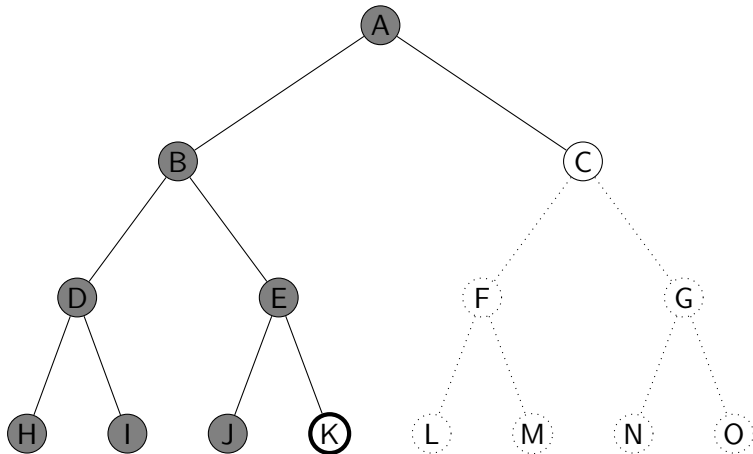
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



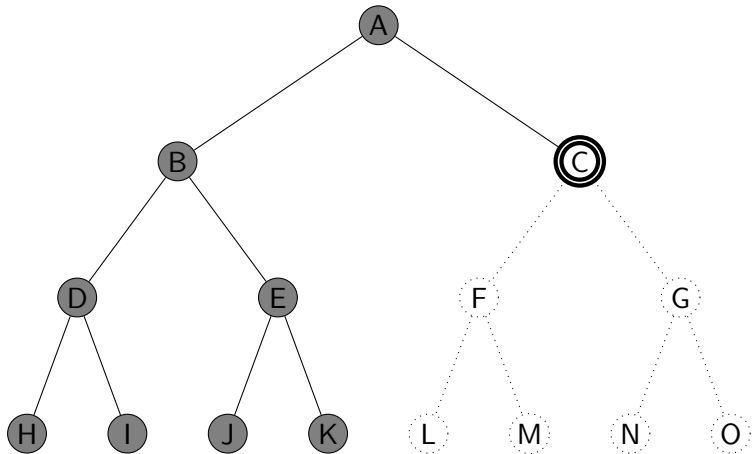
Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



Prohledávání do hloubky

```
function DEPTHFIRSTSEARCH(problem, path ← [])  
  if length(path) = 0 then  
    return DepthFirstSearch(problem, [problem.init_state])  
  current_node ← path.last() # poslední prvek cesty  
  if problem.is_goal(current_node) then  
    print path # cesta k řešení  
  foreach child in problem.moves(current_node) do # možné akce  
    if child ∉ path then # test cyklu v cestě, obecně být nemusí  
      DepthFirstSearch(problem, path + [child]) # rozšířená cesta
```

Prohledávání do hloubky

```
function DEPTHFIRSTSEARCH(problem, path ← [])  
  if length(path) = 0 then  
    return DepthFirstSearch(problem, [problem.init_state])  
  current_node ← path.last() # poslední prvek cesty  
  if problem.is_goal(current_node) then  
    print path # cesta k řešení  
  foreach child in problem.moves(current_node) do # možné akce  
    if child ∉ path then # test cyklu v cestě, obecně být nemusí  
      DepthFirstSearch(problem, path + [child]) # rozšířená cesta
```

úplnost

optimálnost

časová složitost

prostorová složitost

Prohledávání do hloubky

```
function DEPTHFIRSTSEARCH(problem, path ← [])  
  if length(path) = 0 then  
    return DepthFirstSearch(problem, [problem.init_state])  
  current_node ← path.last() # poslední prvek cesty  
  if problem.is_goal(current_node) then  
    print path # cesta k řešení  
  foreach child in problem.moves(current_node) do # možné akce  
    if child ∉ path then # test cyklu v cestě, obecně být nemusí  
      DepthFirstSearch(problem, path + [child]) # rozšířená cesta
```

úplnost **není** úplný (nekonečná větev, cykly)

optimálnost

časová složitost

prostorová složitost

Prohledávání do hloubky

```
function DEPTHFIRSTSEARCH(problem, path ← [])  
  if length(path) = 0 then  
    return DepthFirstSearch(problem, [problem.init_state])  
  current_node ← path.last() # poslední prvek cesty  
  if problem.is_goal(current_node) then  
    print path # cesta k řešení  
  foreach child in problem.moves(current_node) do # možné akce  
    if child ∉ path then # test cyklu v cestě, obecně být nemusí  
      DepthFirstSearch(problem, path + [child]) # rozšířená cesta
```

úplnost **není** úplný (nekonečná větev, cykly)

optimálnost **není** optimální

časová složitost

prostorová složitost

Prohledávání do hloubky

```
function DEPTHFIRSTSEARCH(problem, path ← [])  
  if length(path) = 0 then  
    return DepthFirstSearch(problem, [problem.init_state])  
  current_node ← path.last() # poslední prvek cesty  
  if problem.is_goal(current_node) then  
    print path # cesta k řešení  
  foreach child in problem.moves(current_node) do # možné akce  
    if child ∉ path then # test cyklu v cestě, obecně být nemusí  
      DepthFirstSearch(problem, path + [child]) # rozšířená cesta
```

úplnost	není úplný (nekonečná větev, cykly)
optimálnost	není optimální
časová složitost	$O(b^m)$
prostorová složitost	

Prohledávání do hloubky

```
function DEPTHFIRSTSEARCH(problem, path ← [])  
  if length(path) = 0 then  
    return DepthFirstSearch(problem, [problem.init_state])  
  current_node ← path.last() # poslední prvek cesty  
  if problem.is_goal(current_node) then  
    print path # cesta k řešení  
  foreach child in problem.moves(current_node) do # možné akce  
    if child ∉ path then # test cyklu v cestě, obecně být nemusí  
      DepthFirstSearch(problem, path + [child]) # rozšířená cesta
```

úplnost	není úplný (nekonečná větev, cykly)
optimálnost	není optimální
časová složitost	$O(b^m)$
prostorová složitost	$O(bm)$, lineární

Prohledávání do hloubky

```

function DEPTHFIRSTSEARCH(problem, path ← [])
  if length(path) = 0 then
    return DepthFirstSearch(problem, [problem.init_state])
  current_node ← path.last() # poslední prvek cesty
  if problem.is_goal(current_node) then
    print path # cesta k řešení
  foreach child in problem.moves(current_node) do # možné akce
    if child ∉ path then # test cyklu v cestě, obecně být nemusí
      DepthFirstSearch(problem, path + [child]) # rozšířená cesta

```

úplnost není úplný (nekonečná větev, cykly)

optimálnost není optimální

časová složitost $O(b^m)$

prostorová složitost $O(bm)$, lineární

Největší problém – **nekonečná větev** = nenajde se cíl, program neskončí!

Prohledávání do hloubky s limitem

Řešení nekonečné větve – použití “zarážky” = limit hloubky ℓ

Prohledávání do hloubky s limitem

Řešení nekonečné větve – použití “zarážky” = limit hloubky ℓ

konec má dvě možné interpretace – vyčerpání limitu nebo neexistenci (dalšího) řešení

Prohledávání do hloubky s limitem

Řešení nekonečné větve – použití “zarážky” = limit hloubky ℓ

konec má dvě možné interpretace – **vyčerpání limitu** nebo **neexistenci (dalšího) řešení**

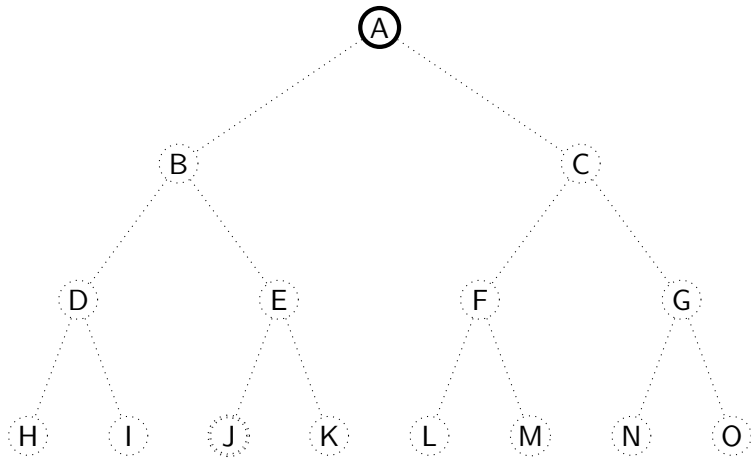
Vlastnosti:

<i>úplnost</i>	není úplný (pro $\ell < d$)
<i>optimálnost</i>	není optimální (pro $\ell > d$)
<i>časová složitost</i>	$O(b^\ell)$
<i>prostorová složitost</i>	$O(b\ell)$

dobrá volba limitu ℓ – podle znalosti problému

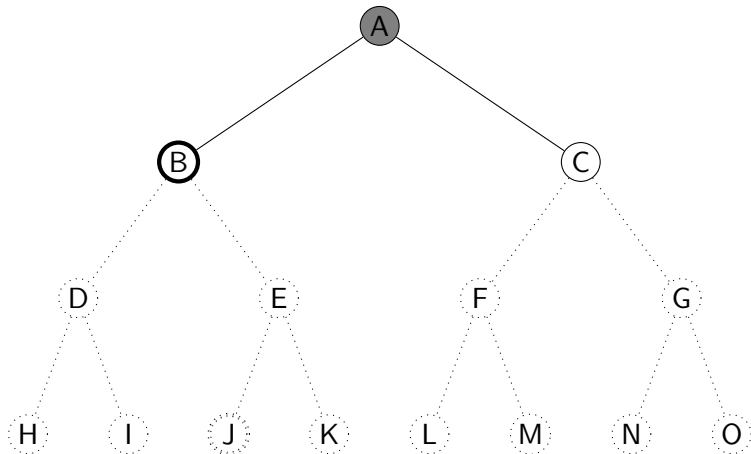
Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



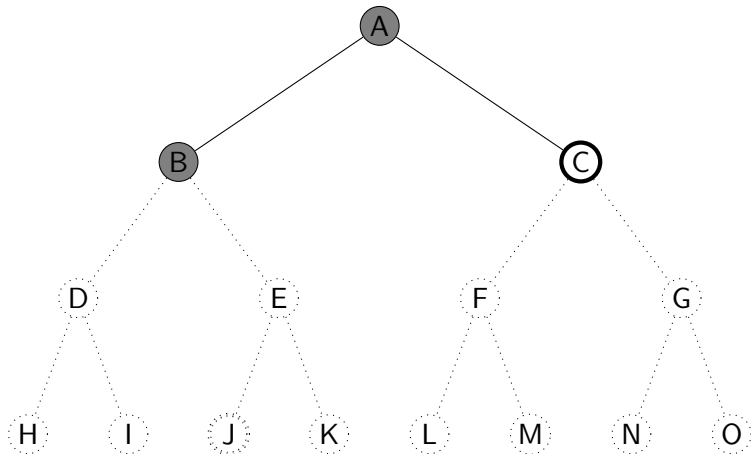
Hledání do šířky

Hledá se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



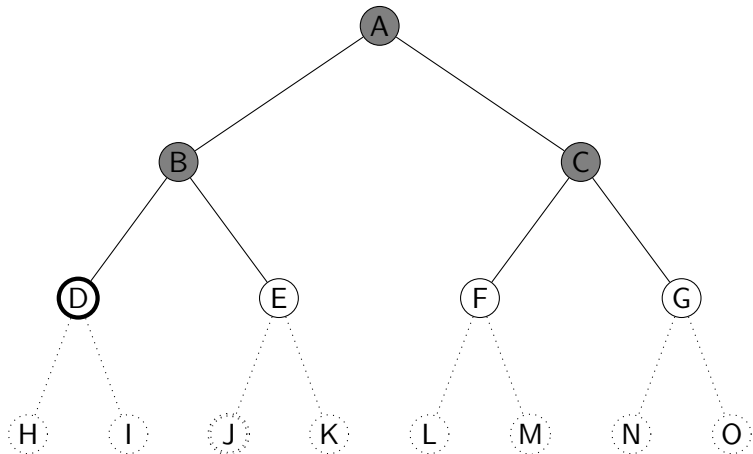
Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



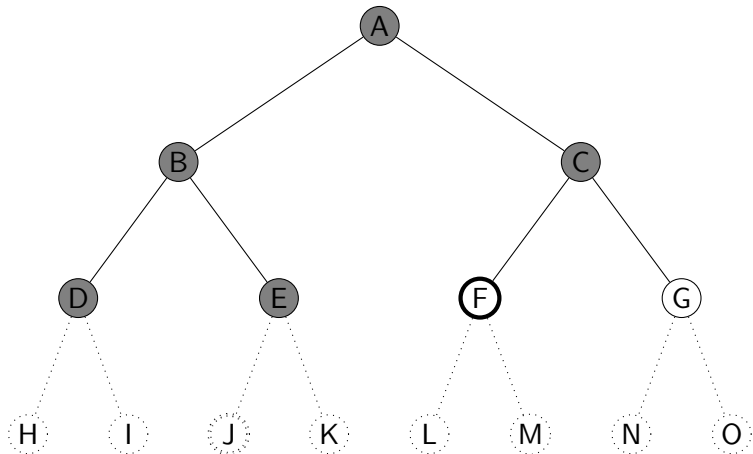
Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



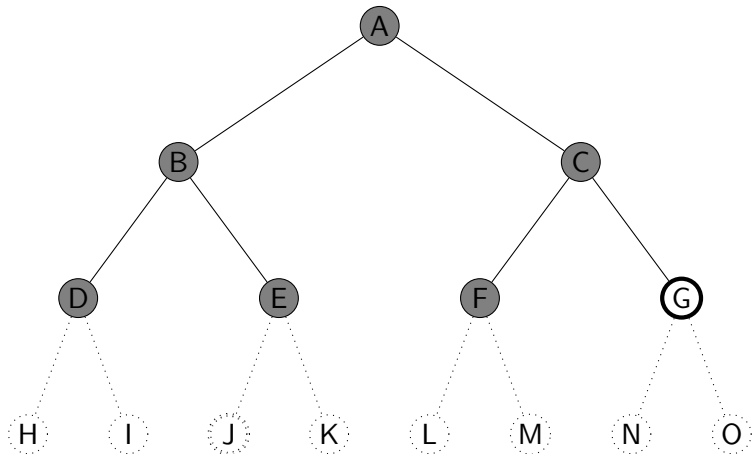
Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



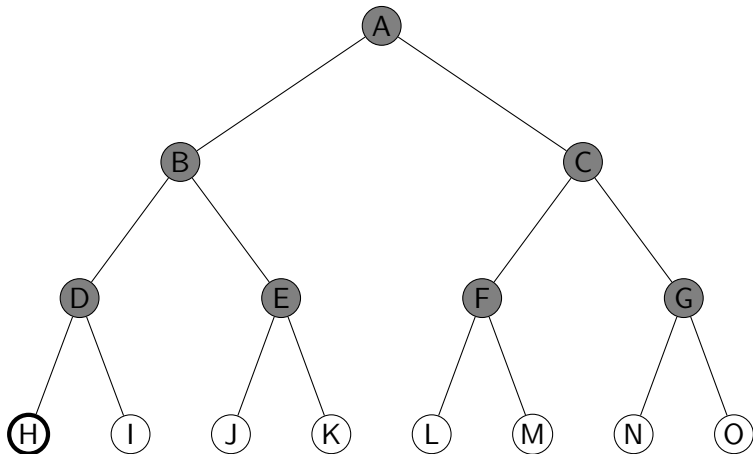
Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



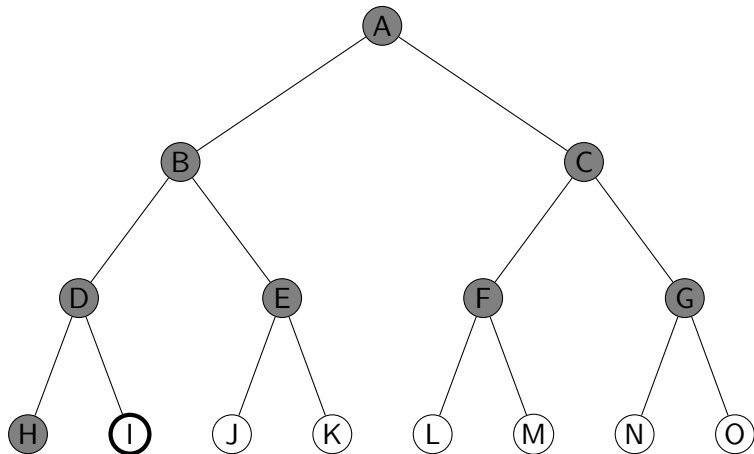
Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



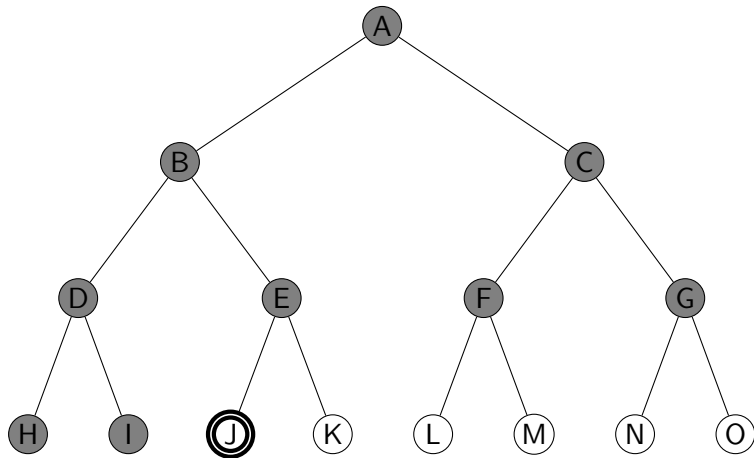
Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



Prohledávání do šířky

ve frontě (FIFO) udržuje seznam cest

```
function BREADTHFIRSTSEARCH(problem)
  process ← [[problem.init_state]] # seznam cest k aktuálnímu uzlu
  while length(process) > 0 do
    current_path ← process.first () # aktuální cesta, první v kolekci
    current_node ← current_path.last() # aktuální uzel, poslední v cestě
    if problem.is_goal(current_node) then
      print current_path # vypíše cestu k řešení
    foreach child in problem.moves(current_node) do
      if child ∉ current_path then # test cyklu v cestě, obecně být nemusí
        process ← process + [current_path + [child]] # rozšířený seznam cest
```

Prohledávání do šířky – vlastnosti

úplnost

optimálnost

časová složitost

prostorová složitost

Prohledávání do šířky – vlastnosti

úplnost je úplný (pro konečné b)

optimálnost

časová složitost

prostorová složitost

Prohledávání do šířky – vlastnosti

úplnost

je úplný (pro konečné b)

optimálnost

je optimální podle délky cesty/**není** optimální podle obecné ceny

časová složitost

prostorová složitost

Prohledávání do šířky – vlastnosti

<i>úplnost</i>	je úplný (pro konečné b)
<i>optimálnost</i>	je optimální podle délky cesty/ není optimální podle obecné ceny
<i>časová složitost</i>	$1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, exponenciální v d
<i>prostorová složitost</i>	

Prohledávání do šířky – vlastnosti

<i>úplnost</i>	je úplný (pro konečné b)
<i>optimálnost</i>	je optimální podle délky cesty/ není optimální podle obecné ceny
<i>časová složitost</i>	$1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, exponenciální v d
<i>prostorová složitost</i>	$O(b^{d+1})$ (každý uzel v paměti)

Prohledávání do šířky – vlastnosti

<i>úplnost</i>	je úplný (pro konečné b)
<i>optimálnost</i>	je optimální podle délky cesty/ není optimální podle obecné ceny
<i>časová složitost</i>	$1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, exponenciální v d
<i>prostorová složitost</i>	$O(b^{d+1})$ (každý uzel v paměti)

Největší problém – paměť:

Hloubka	Uzlů	Čas	Paměť
2	1100	0.11 sek	1 MB
4	111 100	11 sek	106 MB
6	10^7	19 min	10 GB
8	10^9	31 hod	1 TB
10	10^{11}	129 dnů	101 TB
12	10^{13}	35 let	10 PB
14	10^{15}	3 523 let	1 EB

Ani čas není dobrý → potřebujeme **informované** strategie prohledávání.

Prohledávání podle ceny

- BFS je optimální pro rovnoměrně ohodnocené stromy × **prohledávání podle ceny** (**Uniform-cost Search**) je optimální pro **obecné ohodnocení**
- fronta uzlů se udržuje **uspořádaná** podle ceny cesty

Prohledávání podle ceny

- BFS je optimální pro rovnoměrně ohodnocené stromy × **prohledávání podle ceny** (**Uniform-cost Search**) je optimální pro **obecné ohodnocení**
- fronta uzlů se udržuje **uspořádaná** podle ceny cesty

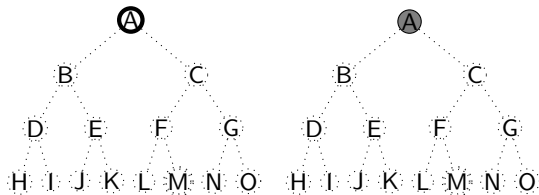
Vlastnosti:

<i>úplnost</i>	je úplný (pro $\epsilon \geq 0$ a b konečné)
<i>optimálnost</i>	je optimální (pro $g(n)$ rostoucí)
<i>časová složitost</i>	počet uzlů s $g \leq C^*$, $O(b^{1+\lceil C^*/\epsilon \rceil})$, kde C^* ... cena optimálního řešení
<i>prostorová složitost</i>	počet uzlů s $g \leq C^*$, $O(b^{1+\lceil C^*/\epsilon \rceil})$

Prohledávání s postupným prohlubováním

prohledávání do hloubky s postupně se **zvyšujícím limitem** (**Iterative deepening DFS, IDS**)

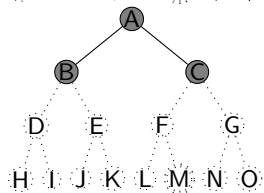
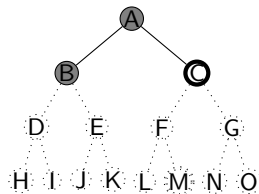
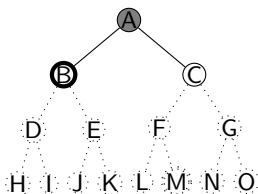
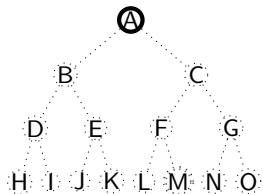
limit=0



Prohledávání s postupným prohlubováním

prohledávání do hloubky s postupně se **zvyšujícím limitem** (**Iterative deepening DFS, IDS**)

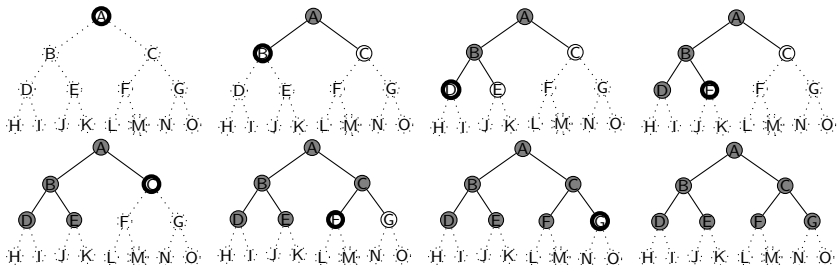
limit=1



Prohledávání s postupným prohlubováním

prohledávání do hloubky s postupně se **zvyšujícím limitem** (**Iterative deepening DFS, IDS**)

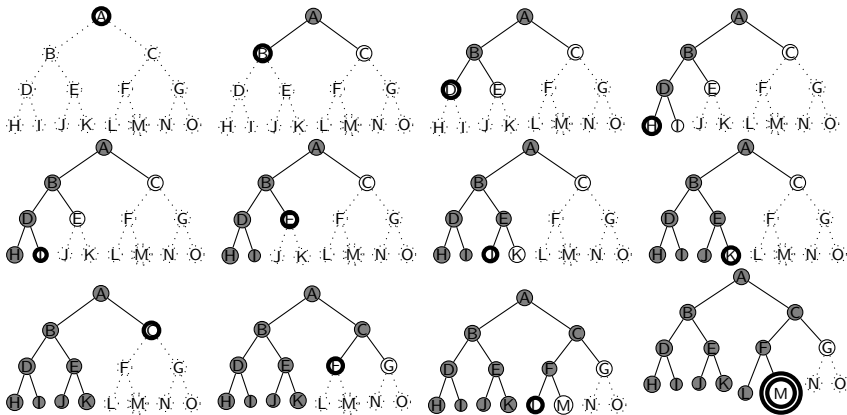
limit=2



Prohledávání s postupným prohlubováním

prohledávání do hloubky s postupně se **zvyšujícím limitem** (**Iterative deepening DFS, IDS**)

limit=3



Prohledávání s postupným prohlubováním – vlastnosti

<i>úplnost</i>	je úplný (pro konečné b)
<i>optimálnost</i>	je optimální (pro $g(n)$ rovnoměrně neklesající funkce hloubky)
<i>časová složitost</i>	$d(b) + (d - 1)b^2 + \dots + 1(b^d) = O(b^d)$
<i>prostorová složitost</i>	$O(bd)$

Prohledávání s postupným prohlubováním – vlastnosti

úplnost je úplný (pro konečné b)

optimálnost je optimální (pro $g(n)$ rovnoměrně neklesající funkce hloubky)

časová složitost $d(b) + (d - 1)b^2 + \dots + 1(b^d) = O(b^d)$

prostorová složitost $O(bd)$

- kombinuje výhody BFS a DFS:
 - nízké paměťové nároky – **lineární**
 - **optimálnost**, **úplnost**

Prohledávání s postupným prohlubováním – vlastnosti

<i>úplnost</i>	je úplný (pro konečné b)
<i>optimálnost</i>	je optimální (pro $g(n)$ rovnoměrně neklesající funkce hloubky)
<i>časová složitost</i>	$d(b) + (d - 1)b^2 + \dots + 1(b^d) = O(b^d)$
<i>prostorová složitost</i>	$O(bd)$

- kombinuje výhody BFS a DFS:
 - nízké paměťové nároky – **lineární**
 - **optimálnost**, **úplnost**
- **zdánlivé plýtvání** opakovaným generováním

ALE **generuje** o jednu úroveň **míň**, např. pro $b = 10$, $d = 5$:

$$N(\text{IDS}) = 50 + 400 + 3\,000 + 20\,000 + 100\,000 = 123\,450$$

$$N(\text{BFS}) = 10 + 100 + 1\,000 + 10\,000 + 100\,000 + 999\,990 = 1\,111\,100$$

Prohledávání s postupným prohlubováním – vlastnosti

úplnost je úplný (pro konečné b)

optimálnost je optimální (pro $g(n)$ rovnoměrně neklesající funkce hloubky)

časová složitost $d(b) + (d - 1)b^2 + \dots + 1(b^d) = O(b^d)$

prostorová složitost $O(bd)$

- kombinuje výhody BFS a DFS:
 - nízké paměťové nároky – **lineární**
 - **optimálnost**, **úplnost**
- **zdánlivé plýtvání** opakovaným generováním

ALE **generuje** o jednu úroveň **míň**, např. pro $b = 10, d = 5$:

$$N(\text{IDS}) = 50 + 400 + 3\,000 + 20\,000 + 100\,000 = 123\,450$$

$$N(\text{BFS}) = 10 + 100 + 1\,000 + 10\,000 + 100\,000 + 999\,990 = 1\,111\,100$$

IDS je **nejvhodnější** neinformovaná strategie pro **velké prostory** a **neznámou hloubku** řešení.

Shrnutí vlastností algoritmů neinformovaného prohledávání

<i>Vlastnost</i>	<i>do hloubky</i>	<i>do hloubky s limitem</i>	<i>do šířky</i>	<i>podle ceny</i>	<i>s postupným prohlubováním</i>
<i>úplnost</i>	ne	ano, pro $\ell \geq d$	ano*	ano*	ano*
<i>optimálnost</i>	ne	ne	ano*	ano*	ano*
<i>časová složitost</i>	$O(b^m)$	$O(b^\ell)$	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^d)$
<i>prostorová složitost</i>	$O(bm)$	$O(b\ell)$	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bd)$

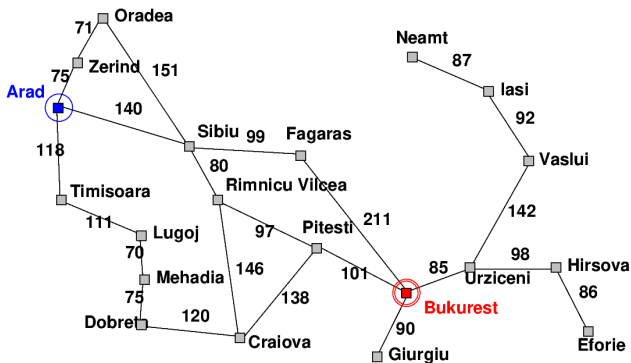


Obsah

- 1 Prohledávání stavového prostoru
 - Řešení problému prohledáváním
 - Prohledávací strategie
- 2 Neinformované prohledávání
 - Prohledávání do hloubky
 - Prohledávání do hloubky s limitem
 - Prohledávání do šířky
 - Prohledávání podle ceny
 - Prohledávání s postupným prohlubováním
- 3 Informované prohledávání stavového prostoru
 - Hladové heuristické hledání
 - Hledání nejlepší cesty – algoritmus A^*
- 4 Jak najít dobrou heuristiku?
 - Jak najít přípustnou heuristickou funkci?
 - Určení kvality heuristiky

Příklad – cesta na mapě

Najdi cestu z města **Arad** do města **Bukurest**



Příklad – schéma rumunských měst

Města:

Arad

Bukurest

Craiova

Dobreta

Eforie

Fagaras

Giurgiu

Hirsova

Iasi

Lugoj

Mehadia

Neamt

...

Cesty:

Arad ↔ Timisoara 118

Arad ↔ Sibiu 140

Arad ↔ Zerind 75

Timisoara ↔ Lugoj 111

Sibiu ↔ Fagaras 99

Sibiu ↔ Rimnicu Vilcea 80

Zerind ↔ Oradea 71

... ↔ ...

Giurgiu ↔ **Bukurest** 90

Pitesti ↔ **Bukurest** 101

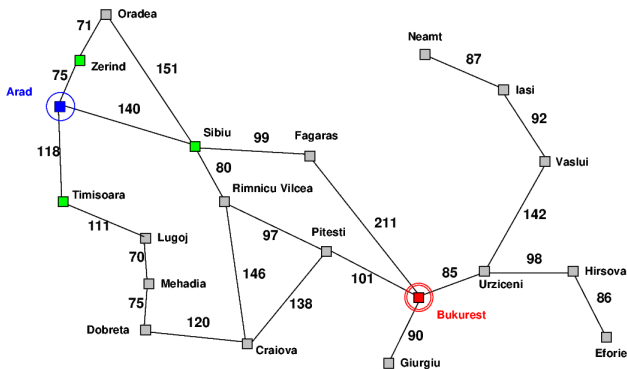
Fagaras ↔ **Bukurest** 211

Urziceni ↔ **Bukurest** 85

Příklad – cesta na mapě

Najdi cestu z města **Arad** do města **Bukurest**

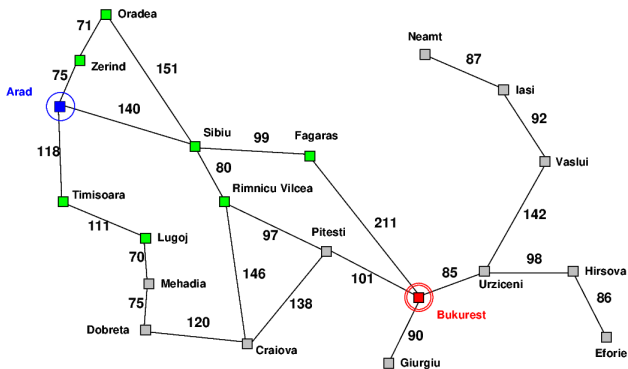
výběr uzlů pomocí **BFS**:



Příklad – cesta na mapě

Najdi cestu z města **Arad** do města **Bukurest**

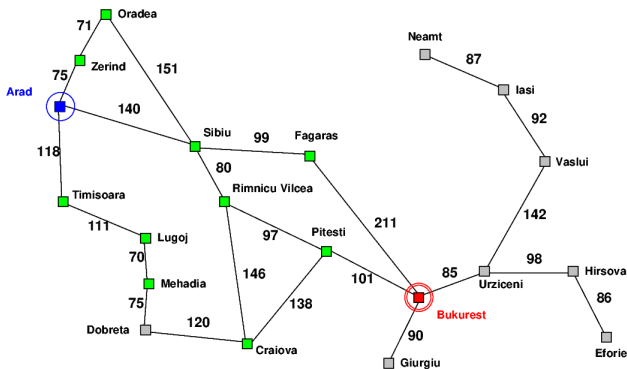
výběr uzlů pomocí **BFS**:



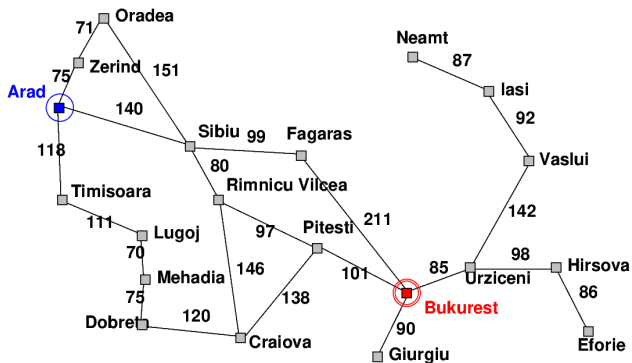
Příklad – cesta na mapě

Najdi cestu z města **Arad** do města **Bukurest**

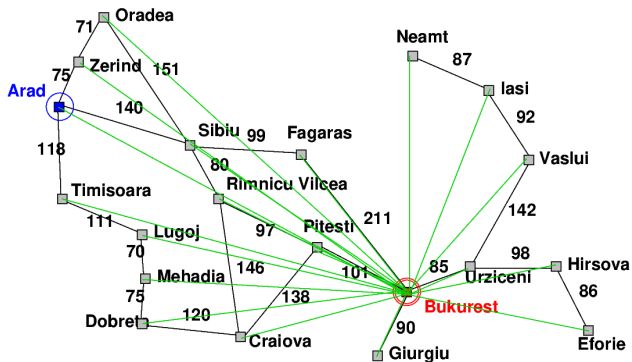
výběr uzlů pomocí **BFS**:



Příklad – schéma rumunských měst



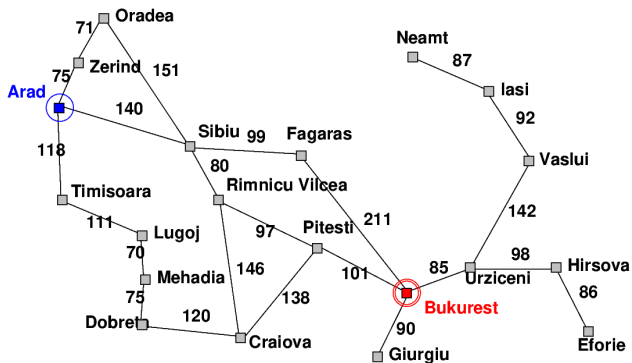
Příklad – schéma rumunských měst



Arad	366
Bukurest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Příklad – schéma rumunských měst



Arad	366
Bukurest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Příklad – cesta na mapě

Neinformované prohledávání:

- DFS, BFS a varianty
- nemá (téměř) žádné informace o pozici cíle – **slepé prohledávání**
- zná pouze:
 - počáteční/cílový stav
 - přechodovou funkci

Příklad – cesta na mapě

Neinformované prohledávání:

- DFS, BFS a varianty
- nemá (téměř) žádné informace o pozici cíle – **slepé prohledávání**
- zná pouze:
 - počáteční/cílový stav
 - přechodovou funkci

Informované prohledávání:

má navíc informaci o (odhadu) blízkosti stavu k cílovému stavu – **heuristická funkce** (heuristika)

Heuristické hledání nejlepší cesty

- Best-first Search (obecný, varianty podle ohodnocovací funkce)
- použití **ohodnocovací funkce** $f(n)$ pro každý uzel – výpočet **přínosu** daného uzlu (stavu)
- udržujeme seznam uzlů **uspořádaný** (vzestupně) vzhledem k $f(n)$
- použití **heuristické funkce** $h(n)$ pro každý uzel – **odhad vzdálenosti** daného uzlu (stavu) od cíle
- čím **menší** $h(n)$, tím **blíže** k cíli, $h(\text{Goal}) = 0$.
- nejjednodušší varianta –
hladové heuristické hledání, *Greedy best-first search*
 $f(n) = h(n)$

Hladové heuristické hledání – příklad

Hledání cesty z města *Arad* do města *Bukurest*

ohodnocovací funkce $f(n) = h(n) = h_{\text{vzd_Buk}}(n)$, **přímá vzdálenost** z n do Bukuresti

Arad

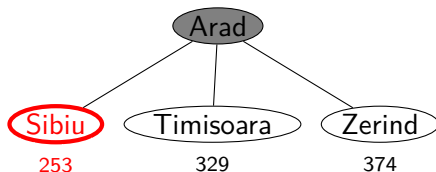
366



Hledové heuristické hledání – příklad

Hledání cesty z města *Arad* do města *Bukurest*

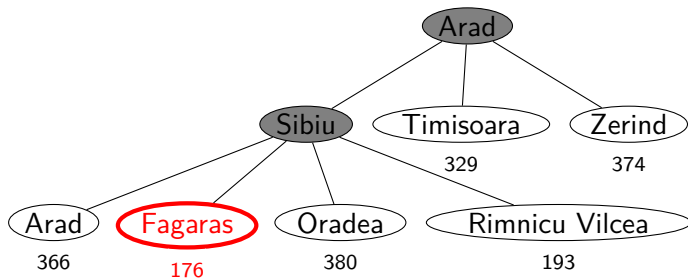
ohodnocovací funkce $f(n) = h(n) = h_{\text{vzd_Buk}}(n)$, **přímá vzdálenost** z n do Bukuresti



Hladové heuristické hledání – příklad

Hledání cesty z města *Arad* do města *Bukurest*

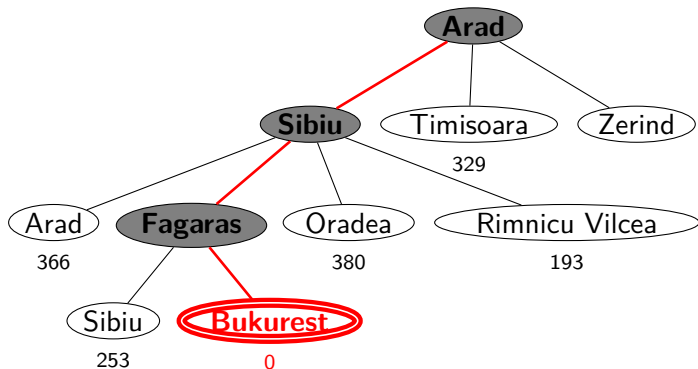
ohodnocovací funkce $f(n) = h(n) = h_{\text{vzd_Buk}}(n)$, **přímá vzdálenost** z n do Bukuresti



Hladové heuristické hledání – příklad

Hledání cesty z města *Arad* do města *Bukurest*

ohodnocovací funkce $f(n) = h(n) = h_{\text{vzd_Buk}}(n)$, **přímá vzdálenost** z n do Bukuresti



Hladové heuristické hledání – vlastnosti

- expanduje vždy uzel, který **se zdá** nejbližší k cíli
- cesta nalezená v příkladu ($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bukurest}) = 450$) je sice úspěšná, ale **není optimální**
($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{RimnicuVilcea} \rightarrow \text{Pitesti} \rightarrow \text{Bukurest}) = 418$)
- *úplnost*
optimálnost
časová složitost
prostorová složitost

Hladové heuristické hledání – vlastnosti

- expanduje vždy uzel, který **se zdá** nejbližší k cíli
- cesta nalezená v příkladu ($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bukurest}) = 450$) je sice úspěšná, ale **není optimální**
($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{RimnicuVilcea} \rightarrow \text{Pitesti} \rightarrow \text{Bukurest}) = 418$)
- *úplnost* obecně **není** úplný (nekonečný prostor, cykly)
optimálnost **není** optimální
časová složitost $O(b^m)$, hodně záleží na h
prostorová složitost $O(b^m)$, každý uzel v paměti

Hledání nejlepší cesty – algoritmus A*

- některé zdroje označují tuto variantu jako **Best-first Search**
- **ohodnocovací funkce** – kombinace $g(n)$ a $h(n)$:

$$f(n) = g(n) + h(n)$$

$g(n)$ je **cena cesty** do n

$h(n)$ je **odhad ceny** cesty z n do cíle

$f(n)$ je **odhad ceny nejlevnější cesty**, která vede přes n

Hledání nejlepší cesty – algoritmus A*

- některé zdroje označují tuto variantu jako **Best-first Search**
- **ohodnocovací funkce** – kombinace $g(n)$ a $h(n)$:

$$f(n) = g(n) + h(n)$$

$g(n)$ je **cena cesty** do n

$h(n)$ je **odhad ceny** cesty z n do cíle

$f(n)$ je **odhad ceny nejlevnější cesty**, která vede přes n

- A* algoritmus vyžaduje tzv. **přípustnou** (*admissible*) heuristiku:

$$0 \leq h(n) \leq h^*(n), \text{ kde } h^*(n) \text{ je skutečná cena cesty z } n \text{ do cíle}$$

tj. odhad se volí vždycky **kratší** nebo roven ceně libovolné **možné** cesty do cíle

Např. přímá vzdálenost $h_{\text{vzd_Buk}}$ nikdy není delší než (jakákoliv) cesta

Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*

ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\text{vzd_Buk}}(n)$

Arad

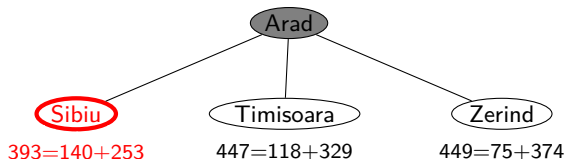
$$366=0+366$$



Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*

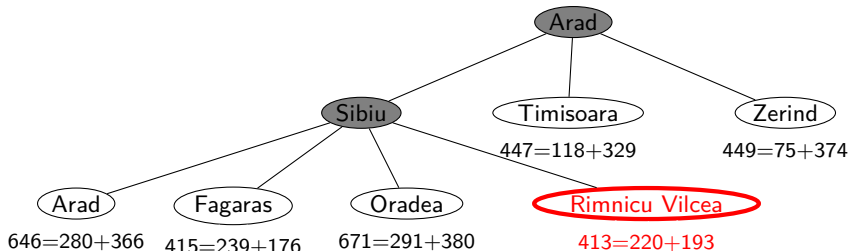
ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\text{vzd_Buk}}(n)$



Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*

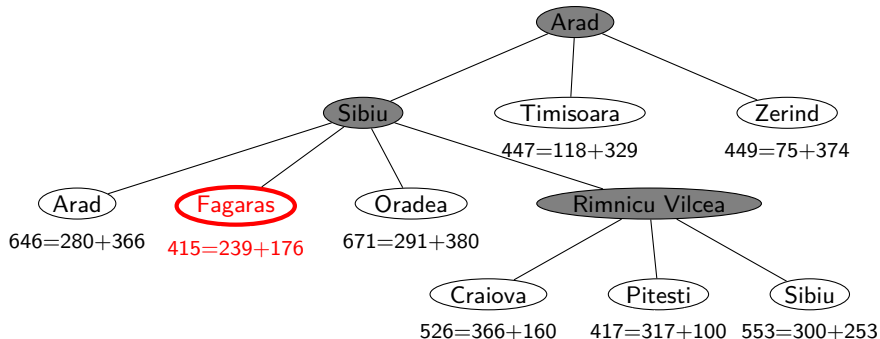
ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\text{vzd_Buk}}(n)$



Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*

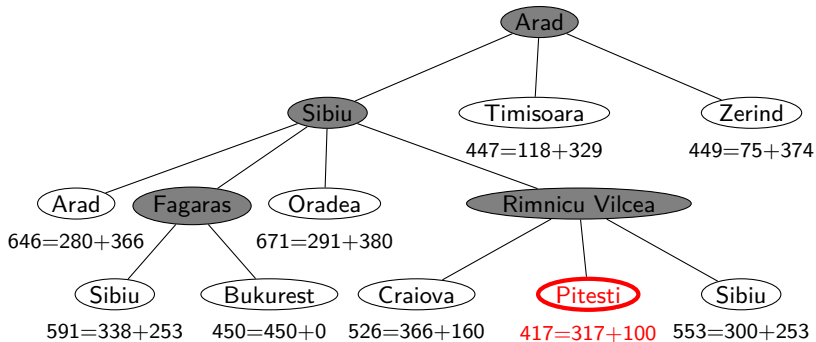
ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\text{vzd_Buk}}(n)$



Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*

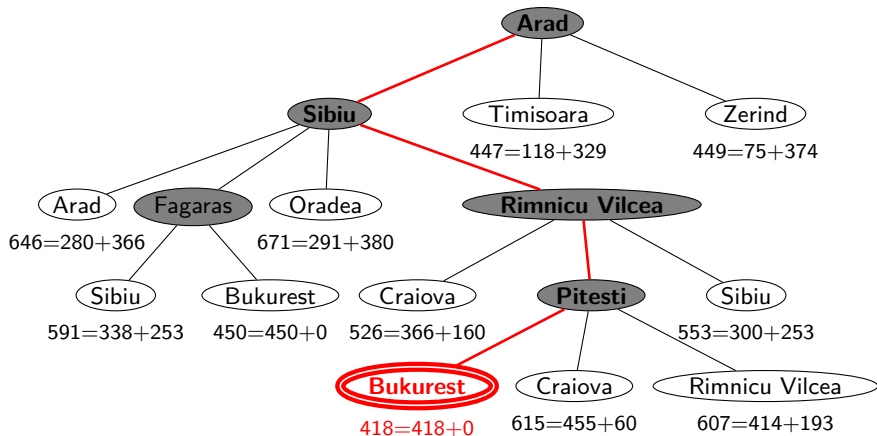
ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\text{vzd_Buk}}(n)$



Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*

ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\text{vzd_Buk}}(n)$



Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$
 - A* expanduje **všechny** uzly s $f(n) < C^*$
 - A* expanduje **některé** uzly s $f(n) = C^*$
 - A* **neexpanduje žádné** uzly s $f(n) > C^*$

- *úplnost*

optimálnost

časová složitost

prostorová složitost

Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$
 - A* expanduje **všechny** uzly s $f(n) < C^*$
 - A* expanduje **některé** uzly s $f(n) = C^*$
 - A* **neexpanduje žádné** uzly s $f(n) > C^*$
- *úplnost* je úplný (pokud [počet uzlů s $f < C^*$] $\neq \infty$, tedy cena $\geq \epsilon$ a b konečné)
- optimálnost*
- časová složitost*
- prostorová složitost*

Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$
 - A* expanduje **všechny** uzly s $f(n) < C^*$
 - A* expanduje **některé** uzly s $f(n) = C^*$
 - A* **neexpanduje žádné** uzly s $f(n) > C^*$
- *úplnost* je úplný (pokud [počet uzlů s $f < C^*$] $\neq \infty$, tedy cena $\geq \epsilon$ a b konečné)
- optimálnost* je optimální
- časová složitost*
- prostorová složitost*

Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$
 - A* expanduje **všechny** uzly s $f(n) < C^*$
 - A* expanduje **některé** uzly s $f(n) = C^*$
 - A* **neexpanduje žádné** uzly s $f(n) > C^*$
- *úplnost* je úplný (pokud [počet uzlů s $f < C^*$] $\neq \infty$, tedy cena $\geq \epsilon$ a b konečné)
- optimálnost* je optimální
- časová složitost* $O((b^*)^d)$, exponenciální v délce řešení d
 b^* ... tzv. *efektivní faktor větvení*, viz dále
- prostorová složitost*

Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$
 - A* expanduje **všechny** uzly s $f(n) < C^*$
 - A* expanduje **některé** uzly s $f(n) = C^*$
 - A* **neexpanduje žádné** uzly s $f(n) > C^*$
- *úplnost* je úplný (pokud [počet uzlů s $f < C^*$] $\neq \infty$, tedy cena $\geq \epsilon$ a b konečné)
- optimálnost* je optimální
- časová složitost* $O((b^*)^d)$, exponenciální v délce řešení d
 b^* ... tzv. *efektivní faktor větvení*, viz dále
- prostorová složitost* $O((b^*)^d)$, každý uzel v paměti

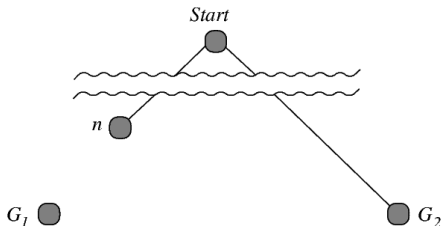
Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$
 - A* expanduje **všechny** uzly s $f(n) < C^*$
 - A* expanduje **některé** uzly s $f(n) = C^*$
 - A* **neexpanduje žádné** uzly s $f(n) > C^*$
- *úplnost* je úplný (pokud [počet uzlů s $f < C^*$] $\neq \infty$, tedy cena $\geq \epsilon$ a b konečné)
- optimálnost* je optimální
- časová složitost* $O((b^*)^d)$, exponenciální v délce řešení d
 b^* ... tzv. *efektivní faktor větvení*, viz dále
- prostorová složitost* $O((b^*)^d)$, každý uzel v paměti

Problém s prostorovou složitostí řeší algoritmy jako *IDA**, *RBFS*

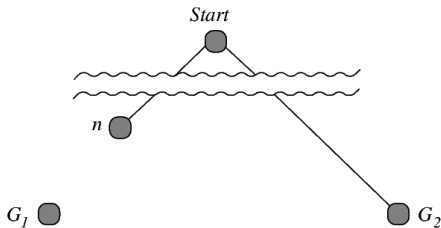
Důkaz optimálnosti algoritmu A*

- předpokládejme, že byl vygenerován nějaký **suboptimální cíl** G_2 a je uložen ve frontě.
- dále necht' n je **neexpandovaný** uzel na nejkratší cestě k **optimálnímu cíli** G_1 (tj. **chybně neexpandovaný** uzel ve správném řešení)



Důkaz optimálnosti algoritmu A*

- předpokládejme, že byl vygenerován nějaký **suboptimální cíl** G_2 a je uložen ve frontě.
- dále necht' n je **neexpandovaný** uzel na nejkratší cestě k **optimálnímu cíli** G_1 (tj. **chybně neexpandovaný** uzel ve správném řešení)

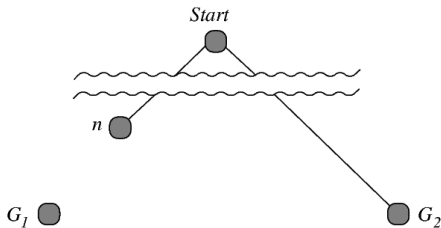


Pak

$$f(G_2) = g(G_2) \quad \text{protože} \quad h(G_2) = 0$$

Důkaz optimálnosti algoritmu A*

- předpokládejme, že byl vygenerován nějaký **suboptimální cíl** G_2 a je uložen ve frontě.
- dále necht' n je **neexpandovaný** uzel na nejkratší cestě k **optimálnímu cíli** G_1 (tj. **chybně neexpandovaný** uzel ve správném řešení)

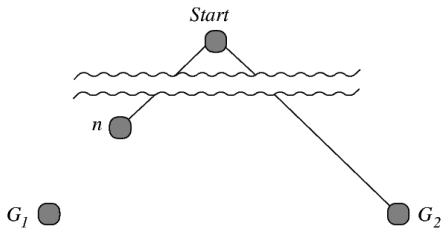


Pak

$$\begin{aligned}
 f(G_2) &= g(G_2) \quad \text{protože } h(G_2) = 0 \\
 &> g(G_1) \quad \text{protože } G_2 \text{ je suboptimální}
 \end{aligned}$$

Důkaz optimálnosti algoritmu A*

- předpokládejme, že byl vygenerován nějaký **suboptimální cíl** G_2 a je uložen ve frontě.
- dále necht' n je **neexpandovaný** uzel na nejkratší cestě k **optimálnímu cíli** G_1 (tj. **chybně neexpandovaný** uzel ve správném řešení)

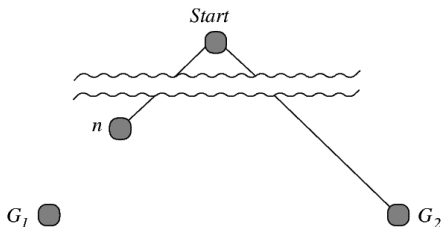


Pak

$$\begin{aligned}
 f(G_2) &= g(G_2) && \text{protože } h(G_2) = 0 \\
 &> g(G_1) && \text{protože } G_2 \text{ je suboptimální} \\
 &\geq f(n) && \text{protože } h \text{ je přípustná}
 \end{aligned}$$

Důkaz optimálnosti algoritmu A*

- předpokládejme, že byl vygenerován nějaký **suboptimální cíl** G_2 a je uložen ve frontě.
- dále necht' n je **neexpandovaný** uzel na nejkratší cestě k **optimálnímu cíli** G_1 (tj. **chybně neexpandovaný** uzel ve správném řešení)



Pak

$$\begin{aligned}
 f(G_2) &= g(G_2) \quad \text{protože } h(G_2) = 0 \\
 &> g(G_1) \quad \text{protože } G_2 \text{ je suboptimální} \\
 &\geq f(n) \quad \text{protože } h \text{ je přípustná}
 \end{aligned}$$

tedy $f(G_2) > f(n) \Rightarrow$ A* nikdy nevybere G_2 pro expanzi dřív než expanduje $n \rightarrow$ **spor** s předpokladem, že n je *neexpandovaný uzel* □

Hledání nejlepší cesty – algoritmus A*

řešení pomocí **prioritní fronty**

```

function A*SEARCH(problem)
  process_heap ← [(0, 0, [problem.init_state])] # prioritní fronta podle f
  while length(process_heap) > 0 do
    f, g, current_path ← process_heap.heap_pop() # nejmenší f-hodnota
    current_node ← current_path.last()
    if problem.is_goal(current_node) then
      print current_path # vypíše cestu k řešení
    foreach child, cost in problem.moves(current_node) do
      if child ∉ current_path then # detekce cyklů, obecně být nemusí
        process_heap.heap_add( (g+cost+h(child),
                               g+cost,
                               current_path + [child]))

```

f-hodnota, g-hodnota a cesta k aktuálnímu uzlu

Příklad – řešení posunovačky

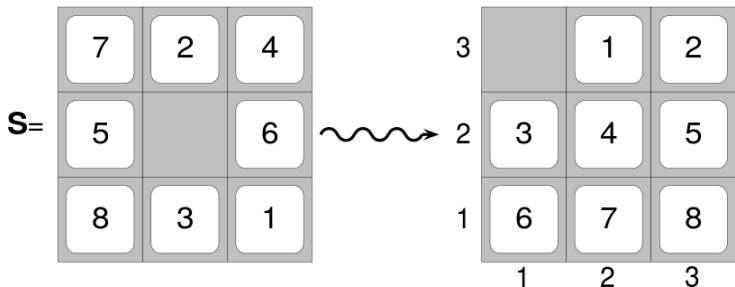
konfigurace = seznam souřadnic (x, y) : [pozicedíry, pozicekámen č.1, ...]

```
8p.init_state ← [(2,2), (3,1), (2,3), (2,1),
(3,3), (1,2), (3,2), (1,3), (1,1)]
```

```
function 8P.IS_GOAL(S)
return S = [(1,3), (2,3), (3,3), (1,2),
(2,2), (3,2), (1,1), (2,1), (3,1)]
```

function 8P.MOVES(state):

- 2 až 4 možnosti – pohyb mezery
- cena vždy 1



Příklad – řešení posunovačky pokrač.

Volba přípustné heuristické funkce h :

Příklad – řešení posunovačky pokrač.

Volba přípustné heuristické funkce h :

- $h_1(n)$ = počet dlaždiček, které nejsou na svém místě $h_1(\mathbf{S}) = 8$

Příklad – řešení posunovačky pokrač.

Volba přípustné heuristické funkce h :

- $h_1(n)$ = počet dlaždiček, které nejsou na svém místě $h_1(\mathbf{S}) = 8$
- $h_2(n)$ = součet **manhattanských vzdáleností** dlaždic od svých správných pozic $h_2(\mathbf{S}) = 3_7 + 1_2 + 2_4 + 2_5 + 3_6 + 2_8 + 2_3 + 3_1 = 18$

Příklad – řešení posunovačky pokrač.

Volba přípustné heuristické funkce h :

- $h_1(n)$ = počet dlaždiček, které nejsou na svém místě $h_1(\mathbf{S}) = 8$
- $h_2(n)$ = součet **manhattanských vzdáleností** dlaždic od svých správných pozic $h_2(\mathbf{S}) = 3_7 + 1_2 + 2_4 + 2_5 + 3_6 + 2_8 + 2_3 + 3_1 = 18$

h_1 i h_2 jsou přípustné ... $h^*(S) = 26$

Příklad – řešení posunovačky pokrač.

Volba přípustné heuristické funkce h :

- $h_1(n)$ = počet dlaždiček, které nejsou na svém místě $h_1(\mathbf{S}) = 8$
- $h_2(n)$ = součet **manhattanských vzdáleností** dlaždic od svých správných pozic $h_2(\mathbf{S}) = 3_7 + 1_2 + 2_4 + 2_5 + 3_6 + 2_8 + 2_3 + 3_1 = 18$

h_1 i h_2 jsou přípustné ... $h^*(S) = 26$

A*Search(8p):

```

[[ (2,2), (3,1), (2,3), (2,1), (3,3), (1,2), (3,2), (1,3), (1,1) ],
 [ (1,2), (3,1), (2,3), (2,1), (3,3), (2,2), (3,2), (1,3), (1,1) ],
 ...
 [ (1,2), (2,3), (3,3), (1,3), (2,2), (3,2), (1,1), (2,1), (3,1) ],
 [ (1,3), (2,3), (3,3), (1,2), (2,2), (3,2), (1,1), (2,1), (3,1) ] ]

```

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak objevit heuristiku h_1 nebo h_2 ?

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak **objevit heuristiku** h_1 nebo h_2 ?
- h_1 i h_2 jsou délky cest pro **zjednodušené verze** problému Posunovačka:
 - při **přenášení** dlaždice kamkoliv – h_1 =počet kroků nejkratšího řešení
 - při **posouvání** dlaždice kamkoliv o **1 pole** (i na plné) – h_2 =počet kroků nejkratšího řešení

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak **objevit heuristiku** h_1 nebo h_2 ?
- h_1 i h_2 jsou délky cest pro **zjednodušené verze** problému Posunovačka:
 - při **přenášení** dlaždice kamkoliv – h_1 =počet kroků nejkratšího řešení
 - při **posouvání** dlaždice kamkoliv o **1 pole** (i na plné) – h_2 =počet kroků nejkratšího řešení
- **relaxovaný problém** – méně omezení na akce než původní problém

*Cena optimálního řešení **relaxovaného problému** je přípustná heuristika pro původní problém.*

optimální řešení **původního** problému = **řešení** relaxovaného problému

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak **objevit heuristiku** h_1 nebo h_2 ?
- h_1 i h_2 jsou délky cest pro **zjednodušené verze** problému Posunovačka:
 - při **přenášení** dlaždice kamkoliv – h_1 =počet kroků nejkratšího řešení
 - při **posouvání** dlaždice kamkoliv o **1 pole** (i na plné) – h_2 =počet kroků nejkratšího řešení

- **relaxovaný problém** – méně omezení na akce než původní problém

Cena optimálního řešení relaxovaného problému je přípustná heuristika pro původní problém.

optimální řešení **původního** problému = **řešení** relaxovaného problému

Posunovačka a relaxovaná posunovačka:

- dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B \wedge B je prázdná

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak **objevit heuristiku** h_1 nebo h_2 ?
- h_1 i h_2 jsou délky cest pro **zjednodušené verze** problému Posunovačka:
 - při **přenášení** dlaždice kamkoliv – h_1 =počet kroků nejkratšího řešení
 - při **posouvání** dlaždice kamkoliv o 1 pole (i na plné) – h_2 =počet kroků nejkratšího řešení
- **relaxovaný problém** – méně omezení na akce než původní problém

*Cena optimálního řešení **relaxovaného problému** je přípustná heuristika pro původní problém.*

optimální řešení **původního** problému = **řešení** relaxovaného problému

Posunovačka a relaxovaná posunovačka:

- dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B \wedge B je prázdná
- (a) dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B
- (b) dlaždice se může přesunout z A na B \Leftrightarrow B je prázdná
- (c) dlaždice se může přesunout z A na B

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak **objevit heuristiku** h_1 nebo h_2 ?
- h_1 i h_2 jsou délky cest pro **zjednodušené verze** problému Posunovačka:
 - při **přenášení** dlaždice kamkoliv – h_1 =počet kroků nejkratšího řešení
 - při **posouvání** dlaždice kamkoliv o 1 pole (i na plné) – h_2 =počet kroků nejkratšího řešení
- **relaxovaný problém** – méně omezení na akce než původní problém

*Cena optimálního řešení **relaxovaného problému** je přípustná heuristika pro původní problém.*

optimální řešení **původního** problému = **řešení** relaxovaného problému

Posunovačka a relaxovaná posunovačka:

- dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B \wedge B je prázdná
- (a) dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B $\dots h_2$
- (b) dlaždice se může přesunout z A na B \Leftrightarrow B je prázdná
- (c) dlaždice se může přesunout z A na B

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak **objevit heuristiku** h_1 nebo h_2 ?
- h_1 i h_2 jsou délky cest pro **zjednodušené verze** problému Posunovačka:
 - při **přenášení** dlaždice kamkoliv – h_1 =počet kroků nejkratšího řešení
 - při **posouvání** dlaždice kamkoliv o 1 pole (i na plné) – h_2 =počet kroků nejkratšího řešení
- **relaxovaný problém** – méně omezení na akce než původní problém

*Cena optimálního řešení **relaxovaného problému** je přípustná heuristika pro původní problém.*

optimální řešení **původního** problému = **řešení** relaxovaného problému

Posunovačka a relaxovaná posunovačka:

- dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B \wedge B je prázdná
- (a) dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B $\dots h_2$
- (b) dlaždice se může přesunout z A na B \Leftrightarrow B je prázdná
- (c) dlaždice se může přesunout z A na B $\dots h_1$

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak **objevit heuristiku** h_1 nebo h_2 ?
- h_1 i h_2 jsou délky cest pro **zjednodušené verze** problému Posunovačka:
 - při **přenášení** dlaždice kamkoliv – h_1 =počet kroků nejkratšího řešení
 - při **posouvání** dlaždice kamkoliv o 1 pole (i na plné) – h_2 =počet kroků nejkratšího řešení
- **relaxovaný problém** – méně omezení na akce než původní problém

*Cena optimálního řešení **relaxovaného problému** je přípustná heuristika pro původní problém.*

optimální řešení **původního** problému = **řešení** relaxovaného problému

Posunovačka a relaxovaná posunovačka:

- dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B \wedge B je prázdná
- (a) dlaždice se může přesunout z A na B \Leftrightarrow A sousedí s B ... h_2
- (b) dlaždice se může přesunout z A na B \Leftrightarrow B je prázdná ... Gaschnigova h.
- (c) dlaždice se může přesunout z A na B h_1

Určení kvality heuristiky

efektivní faktor větvení b^* – N ... počet vygenerovaných uzlů, d ... hloubka řešení, idealizovaný strom s $N + 1$ uzly má faktor větvení b^* (reálné číslo):

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

např.: když A^* najde řešení po 52 uzlech v hloubce 5 ... $b^* = 1.92$
heuristika je tím **lepší**, čím **blíže** je b^* **hodnotě 1**.

Určení kvality heuristiky

efektivní faktor větvení b^* – N ... počet vygenerovaných uzlů, d ... hloubka řešení, idealizovaný strom s $N + 1$ uzly má faktor větvení b^* (reálné číslo):

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

např.: když A* najde řešení po 52 uzlech v hloubce 5 ... $b^* = 1.92$
heuristika je tím **lepší**, čím **blíže** je b^* **hodnotě 1**.

☞ **měření b^*** na množině testovacích sad – dobrá představa o **přínosu heuristiky**

8-posunovačka

d	Průměrný počet uzlů			Efektivní faktor větvení b^*		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
6	680	20	18	2.73	1.34	1.30
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
18	–	3056	363	–	1.46	1.26
24	–	39135	1641	–	1.48	1.26

Určení kvality heuristiky

efektivní faktor větvení b^* – N ... počet vygenerovaných uzlů, d ... hloubka řešení, idealizovaný strom s $N + 1$ uzly má faktor větvení b^* (reálné číslo):

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

např.: když A* najde řešení po 52 uzlech v hloubce 5 ... $b^* = 1.92$
heuristika je tím **lepší**, čím **blíže** je b^* **hodnotě 1**.

☞ **měření b^*** na množině testovacích sad – dobrá představa o **přínosu heuristiky**

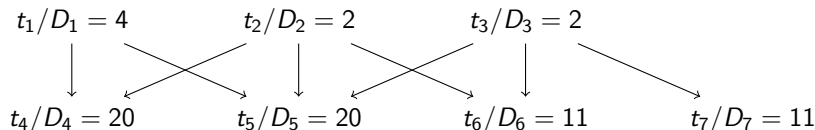
8-posunovačka

d	Průměrný počet uzlů			Efektivní faktor větvení b^*		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
6	680	20	18	2.73	1.34	1.30
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
18	–	3056	363	–	1.46	1.26
24	–	39135	1641	–	1.48	1.26

h_2 **dominuje** h_1 ($\forall n : h_2(n) \geq h_1(n)$) ... h_2 je **lepší** (nebo stejná) než h_1 ve všech případech

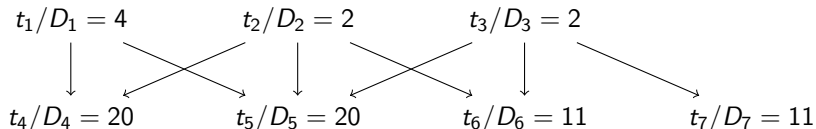
Příklad – rozvrh práce procesorů

- úlohy t_i s potřebným časem na zpracování D_i (např.: $i = 1, \dots, 7$)
- m procesorů (např.: $m = 3$)
- relace **precedence** mezi úlohami – které úlohy mohou začít až po skončení dané úlohy



Příklad – rozvrh práce procesorů

- úlohy t_i s potřebným časem na zpracování D_i (např.: $i = 1, \dots, 7$)
- m procesorů (např.: $m = 3$)
- relace **precedence** mezi úlohami – které úlohy mohou začít až po skončení dané úlohy

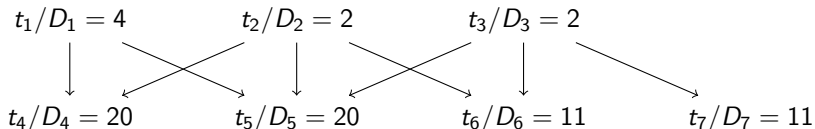


- problém: najít **rozvrh práce** pro každý procesor s minimalizací celkového času

	02	4	13	24	33
CPU ₁	t_3	$\leftarrow t_6 \rightarrow$	$\leftarrow t_5 \rightarrow$		
CPU ₂	t_2	$\leftarrow t_7 \rightarrow$		
CPU ₃	t_1	$\leftarrow t_4 \rightarrow$		

Příklad – rozvrh práce procesorů

- úlohy t_i s potřebným časem na zpracování D_i (např.: $i = 1, \dots, 7$)
- m procesorů (např.: $m = 3$)
- relace **precedence** mezi úlohami – které úlohy mohou začít až po skončení dané úlohy



- problém: najít **rozvrh práce** pro každý procesor s minimalizací celkového času

	02	4	13	24	33
CPU ₁	$t_3 \leftarrow t_6 \Rightarrow$	$\leftarrow t_5 \Rightarrow$			
CPU ₂	$t_2 \leftarrow t_7 \Rightarrow$				
CPU ₃	$t_1 \Rightarrow$	$\leftarrow t_4 \Rightarrow$			

	02	4	13	24	33
CPU ₁	$t_3 \leftarrow t_6 \Rightarrow$	$\leftarrow t_7 \Rightarrow$			
CPU ₂	$t_2 \leftarrow$	$\leftarrow t_5 \Rightarrow$			
CPU ₃	$t_1 \Rightarrow$	$\leftarrow t_4 \Rightarrow$			

Příklad – rozvrh práce procesorů – pokrač.

- stavy: (**nezařazené_úlohy**, **běžící_úlohy**, **čas_ukončení**)

př.: $([(Waiting_{T_1}, D_1), (Waiting_{T_2}, D_2), \dots], [(Task_1, F_1), (Task_2, F_2), (Task_3, F_3)], FinTime)$

běžící_úlohy udržujeme setříděné $F_1 \leq F_2 \leq F_3$

Příklad – rozvrh práce procesorů – pokrač.

- stavy: (**nezařazené_úlohy**, **běžící_úlohy**, **čas_ukončení**)
př.: $([(Waiting_{T_1, D_1}), (Waiting_{T_2, D_2}), \dots], [(Task_1, F_1), (Task_2, F_2), (Task_3, F_3)], FinTime)$
běžící_úlohy udržujeme setříděné $F_1 \leq F_2 \leq F_3$
- počáteční uzel:

```
proc. init_state ← ((("t1",4), ("t2",2), ("t3",2), ("t4",20), ("t5",20), ("t6",11),  
                    ("t7",11)), [("idle",0), ("idle",0), ("idle",0)], 0)
```

Příklad – rozvrh práce procesorů – pokrač.

- stavy: (**nezařazené_úlohy**, **běžící_úlohy**, **čas_ukončení**)
 př.: $([(WaitingT_1, D_1), (WaitingT_2, D_2), \dots], [(Task_1, F_1), (Task_2, F_2), (Task_3, F_3)], FinTime)$
běžící_úlohy udržujeme setříděné $F_1 \leq F_2 \leq F_3$

- počáteční uzel:

```
proc. init_state ← ([("t1",4), ("t2",2), ("t3",2), ("t4",20), ("t5",20), ("t6",11),
                    ("t7",11)], [{"idle",0}, {"idle",0}, {"idle",0}], 0)
```

- přechodová funkce **proc.moves(Stav)** → **nové stavy s cenami**:

```
function PROC.MOVES (state)
    waiting, active, fintime = state
    moves ← []
    for task in waiting do # kontrolujem přednost v nezařazených a nedokončených
        if not check_precedence_waiting(task, waiting.without(task)) then next
        if not check_precedence_active(task, active) then next
        newactive, newfintime ← active.replace_first_and_sort(task)
        moves.append((waiting.without(task), newactive, newfintime))
    moves ← moves + insert_idle(waiting, active, fintime) # čekání na procesor
    return moves
```


Příklad – rozvrh práce procesorů – pokrač.

- stavy: (**nezařazené_úlohy**, **běžící_úlohy**, **čas_ukončení**)
 př.: $([(Waiting_{T_1}, D_1), (Waiting_{T_2}, D_2), \dots], [(Task_1, F_1), (Task_2, F_2), (Task_3, F_3)], FinTime)$
běžící_úlohy udržujeme setříděné $F_1 \leq F_2 \leq F_3$

- počáteční uzel:

```
proc. init_state ← ([("t1",4), ("t2",2), ("t3",2), ("t4",20), ("t5",20), ("t6",11),
                    ("t7",11)], [{"idle",0}, {"idle",0}, {"idle",0}], 0)
```

- přechodová funkce **proc.moves(Stav)** → **nové stavy s cenami**:

```
function PROC.MOVES (state)
    waiting, active, fintime = state
    moves ← []
    for task in waiting do # kontrolujem přednost v nezařazených a nedokončených
        if not check_precedence_waiting(task, waiting.without(task)) then next
        if not check_precedence_active(task, active) then next
        newactive, newfintime ← active.replace_first_and_sort(task)
        moves.append((waiting.without(task), newactive, newfintime))
    moves ← moves + insert_idle(waiting, active, fintime) # čekání na procesor
    return moves
```

- cílová podmínka

```
function PROC.IS_GOAL (state)
    return length(state[1]) = 0 # žádné nezařazené
```

Příklad – rozvrh práce procesorů – pokrač.

- heuristika

Příklad – rozvrh práce procesorů – pokrač.

- heuristika

optimální (nedosažitelný) čas:

$$\mathbf{Finall} = \frac{\sum_i D_i + \sum_j F_j}{m}$$

skutečný (průběžný) čas výpočtu:

$$\mathbf{Fin} = \max(F_j)$$

Příklad – rozvrh práce procesorů – pokrač.

- heuristika

optimální (nedosažitelný) čas:

skutečný (průběžný) čas výpočtu:

$$\mathbf{Finall} = \frac{\sum_i D_i + \sum_j F_j}{m}$$

$$\mathbf{Fin} = \max(F_j)$$

heuristická funkce

$$h = \begin{cases} \mathbf{Finall} - \mathbf{Fin}, & \text{když } \mathbf{Finall} > \mathbf{Fin} \\ 0, & \text{jinak} \end{cases}$$

```
function PROC.H (state)
```

```
tasks, processors, fintime ← state
```

```
total_task_time ← Σ_task_time(tasks) # čas potřebný ke zpracování čekajících úloh
```

```
total_proc_time ← Σ_proc_time(processors) # aktuálně naplánovaný čas všech procesorů
```

```
finall ← (total_task_time + total_proc_time)/length(processors)
```

```
if finall > fintime then
```

```
    return finall - fintime # odhad času pro zpracování čekajících úloh v plánu
```

```
return 0
```

Příklad – rozvrh práce procesorů – pokrač.

A*Search(proc):

```
[
  [(t1,4),(t2,2),(t3,2),(t4,20),(t5,20),(t6,11),(t7,11)], [(idle,0),(idle,0),(idle,0)], 0),
  [(t1,4),(t2,2),(t4,20),(t5,20),(t6,11),(t7,11)], [(idle,0),(idle,0),(t3,2)], 2),
  [(t1,4),(t4,20),(t5,20),(t6,11),(t7,11)], [(idle,0),(t2,2),(t3,2)], 2),
  [(t4,20),(t5,20),(t6,11),(t7,11)], [(t2,2),(t3,2),(t1,4)], 4),
  [(t4,20),(t5,20),(t6,11)], [(t3,2),(t1,4),(t7,13)], 13),
  [(t4,20),(t5,20),(t6,11)], [(idle,4),(t1,4),(t7,13)], 13),
  [(t5,20),(t6,11)], [(t1,4),(t7,13),(t4,24)], 24),
  [(t6,11)], [(t7,13),(t5,24),(t4,24)], 24),
  [], [(t6,24),(t5,24),(t4,24)], 24) ]
```